Early Decision and Stopping in Synchronous Consensus: A Predicate-Based Guided Tour

Armando Castañeda, Yoram Moses, Michel Raynal, Matthieu Roy

Early Decision and Stopping in Synchronous Consensus: a Predicate-Based Guided Tour

Armando Castañeda[1], Yoram Moses[2], Michel Raynal[3,4], Matthieu Roy[5]

[1] Instituto de Matemáticas, UNAM, México
[2] Technion, Haifa, Israel
[3] Institut Universitaire de France
[4] IRISA, Université de Rennes, Rennes, France
[5] LAAS, CNRS, Université de Toulouse, France

Abstract. Consensus is the most basic agreement problem encountered in fault-tolerant distributed computing: each process proposes a value and non-faulty processes must agree on the same value, which has to be one of the proposed values. While this problem is impossible to solve in asynchronous systems prone to process crash failures, it can be solved in synchronous (round-based) systems where all but one process might crash in any execution. It is well-known that $(t + 1)$ rounds are necessary and sufficient in the worst case execution scenario for the processes to decide and stop executing, where $t < n$ is a system parameter denoting the maximum number of allowed process crashes and n denotes the number of processes in the system.
Early decision and stopping considers the case where $f < t$ processes actually crash, f not being known by processes. It has been shown that the number of rounds that have to be executed in the worst case is then $\mathsf{min}(f + 2, t + 1)$. Following Castañeda, Gonczarowski and Moses (DISC 2014), the paper shows that this value is an upper bound attained only in worst execution scenarios. To this end, it investigates a sequence of three early deciding/stopping predicates $P_1 = P_{\mathsf{count}}$, $P_2 = P_{\mathsf{dif}}$ and $P_3 = P_{\mathsf{pref0}}$, of increasing power, which differ in the information obtained by the processes from the actual failure, communication and data pattern. It is shown that each predicate P_i is better than the previous one P_{i-1}, $i \in \{2, 3\}$, in the sense that there are executions where P_i allows processes to reach a decision earlier than P_{i-1}, while P_{i-1} never allows a process to decide earlier than P_i. Moreover, $P_3 = P_{\mathsf{pref0}}$ is an *unbeatable* predicate in the sense that it cannot be strictly improved: if there is an early deciding/stopping predicate P' that improves the decision time of a process with respect to P_{pref0} in a given execution, then there is at least one execution in which a process decides with P' strictly later than with P_{pref0}.

Keywords: Agreement, Consensus, Early decision, Early stopping, Process crash, Round-based algorithm, Synchronous message-passing system, t-Resilience.

1 Introduction

1.1 t-Resilient crash-prone synchronous system

This paper considers a distriuted system with n processes, among which at most t may crash, $1 \leq t < n$. Hence, n and t are two system model parameters that are statically

defined and known when designing an algorithm. A crash is a premature halt: a process behaves correctly, executing the algorithm assigned to it, until it possibly crashes. After a crash, a process executes no more actions. A process that does not crash in a given execution is said to be *correct* or *non-faulty* there, otherwise it is *faulty*. Moreover, given an execution, let f, with $0 \leq f \leq t$, denote the number of processes that actually crash in this execution. Notice that while n and t are two parameters (of the system model) that can be used in an algorithm executed by processes, f is specific to each execution and cannot be known in advance, and consequently no process knows its value.

The processes communicate by broadcasting and receiving messages. If a process does not crash while executing a broadcast, the message is received by all processes, including itself. If it crashes while executing a broadcast, an arbitrary subset of processes (not predetermined and possibly empty) receive the message (without alteration). Hence, a broadcast operation is not atomic.

The processes execute collectively a sequence of synchronous rounds. In each *round* a process first broadcasts a message, then receives messages, and finally executes a local computation whose inputs are its current local state and the messages it has received during the current round. The *synchrony* model assumption states that a message is received in the very same round as the round in which it is sent. Hence, synchrony means that the processes progress in a lock-step manner.

An *distributed algorithm* (or *protocol*) is made of a collection of local algorithms, one per process. Each local algorithm indicates messages to be sent by the corresponding process at each round. Sometimes it is convenient to consider *full-information* algorithms where in every round, each process broadcasts all it knows so far. Full-information algorithms are not meant to be efficient —messages may contain unnecessary information— but are easy to describe and useful to prove lower bounds on step complexity: any information transfer scheme used by another algorithm is contained in the full information transfer scheme.

1.2 The consensus problem

The *consensus* problem was introduced in the Eighties by Lamport, Shostack, and Pease in the context of synchronous message-passing systems prone to Byzantine (arbitrary) failures [14,16]. Here we consider the case of process crash failures.

Each process is assumed to propose a value, and the processes have to agree on the same value. Of course, a process may crash before proposing a value, or before deciding a value. For the problem to be meaningful, the decided value must be related to the proposed values. This is captured by the following properties, which constitute a specification of the consensus problem (hence, any algorithm that claims to solve the problem must satisfy these properties).

- Termination. Every correct process decides on a value.
- Validity. A decided value is a proposed value.
- Agreement. No two (correct or faulty) processes decide different values.

1.3 Bounds on the number of rounds

The bound $(t+1)$. It is shown in [1,10] that $(t+1)$ rounds are necessary and sufficient to solve consensus in a synchronous system prone to up to $t < n$ process crash failures. An intuition that underlies this bound is the following. A "worst case" scenario is when there is a crash per round, which prevents processes from knowing the state of the system at the beginning of the round. But if $(t+1)$ rounds are executed, there is a failure-free round (a.k.a. *clean* round [7]) during which all the correct processes can exchange and obtain proposed values, from which a value can be deterministically extracted to be decided.

The bound $\mathsf{min}(f+2, t+1)$. As t is known by the processes while $f \leq n$ is not, an interesting question is the following: is it possible to solve the consensus problem in crash-prone synchronous systems in fewer than $(t+1)$ rounds when the number of actual crashes f is smaller than t? This question is known as the *early deciding/stopping*, problem [6]. In early stopping, a process stops executing when it decides; In early deciding, a process can continue executing rounds after it has decided. Here, we consider early deciding/stopping algorithms, i.e., algorithms were a process stops executing in the same round as the one in which it decides.

In other words, can we adapt the efficiency of a consensus algorithm to the actual value of f, instead of always having the "$(t+1)$ rounds" cost? Thus, the main target in early deciding/stopping algorithms is to allow at least one process to detect as soon as possible a predicate on the execution, e.g., a failure-free round, which will allow it to safely decide and stop.

It is shown in [2,6,13,18,22] that $\mathsf{min}(f+2, t+1)$ is a necessary and sufficient condition for early deciding/stopping consensus. Interestingly, this bound is independent of the failure model, be it crash failure, omission failure, or Byzantine failure. An intuition for the $(f+2)$ bound is the following. As there are only f failures in the considered execution, after $(f+1)$ rounds there is at least one process that executed a round in which it saw no failures. Thereby, this process knows which value can be decided, but, as $f \neq t$, it does not know if the other processes are aware of it. Hence, it needs an additional round to inform the other processes of this knowledge before deciding.

1.4 Content of the paper

In the following we are interested in predicates that, not only match the lower bound of $\mathsf{min}(f+2, t+1)$ rounds for reaching consensus in worst case scenarios, but allow processes to reach a decision in much fewer rounds in a lot of frequent cases, such as when there are initial crashes, or when several processes crash during the very same round.

These predicates are denoted P_{count}, P_{dif}, and P_{pref0}. We investigate their respective power to solve early deciding/stopping binary consensus[6] and consider those predicates in sequence $P_1 = P_{\mathsf{count}}$, $P_2 = P_{\mathsf{dif}}$ and $P_3 = P_{\mathsf{pref0}}$. We show that each predicate in

[6] While P_{pref0} is specific to binary consensus, P_{count} and P_{dif} can be used for multivalued consensus (where the size of the proposed value is not restricted to be only one bit). However, the predicate can be modified to handle multivalued consensus.

the sequence P_i ($i \in \{2,3\}$) is better than the previous one P_{i-1}: there are executions in which P_i allows processes to reach a decision earlier than P_{i-1}, while P_{i-1} never allows processes to reach a decision earlier than P_i.

To go further, we consider the notion of *unbeatability* [12] (initially called *optimality*), that has been introduced to formally compare the decision-time performance of algorithms. For binary consensus, P_{pref0} is an *unbeatable* predicate in the sense that it cannot be strictly improved: if there is an early deciding/stopping predicate that improves the decision time of a process for binary consensus in a given execution, then there is an execution in which a process decides strictly later than by P_{pref0}. Thus, in principle, there are predicates that can improve the decision time of a process in an execution at the cost of deciding/stopping strictly later in another case.

2 The Three Early Deciding/Stopping Predicates

2.1 P_{count} (P_1) : a Predicate Based on the Counting of Crashed Processes

Let us observe that "to be crashed" is a stable property, i.e., after it crashed, a process never recovers. A crash is a premature halt. This observation can be used to detect process crashes, by requiring each process to broadcast a message at every round, until it decides or crashes. Hence, if r is the first round during which p_i does not receive a message from p_j, and p_i has not yet received a decision message from p_j, then p_i can safely conclude that p_j crashed.

Let $\mathit{faulty}_i[r]$ be the number of processes that p_i considers faulty after the reception of messages during round r, i.e., the number of processes from which it did not receive a message during r. A simple early decision predicate used by p_i at round r is $P_1 = P_{\mathsf{count}}$:

$$P_{\mathsf{count}}[i,r] \equiv \big(\mathit{faulty}_i[r] < r\big).$$

This predicate (used in [17]) specifically targets the worst case scenario: it allows a process p_i to detect the first round in which, from its point of view, there is no crash. Let r be the first round such that $P_{\mathsf{count}}[i,r]$ is true. This means that (a) for any round $r' < r$ we have $\mathit{faulty}_i[r'] \geq r'$, and (b) r is a failure-free round from p_i point of view. Those properties will be exploited to obtain a P_{count}-based early stopping consensus algorithm, that we describe in Section 3.3.

2.2 P_{dif} (P_2): a Round-based Differential Predicate

A second early stopping predicate, introduced in [19], is a differential predicate, in the sense that it is based on each pair of consecutive rounds (the current and the previous rounds). It requires that each process broadcasts a message until it decides or crashes, and each message m indicates if its sender is about to decide after having broadcast m.

Let $UP[r]$ be the set of processes that start round r, i.e., the set of alive processes when round r starts. Let $rec_i[r]$ be the set of processes from which p_i receives messages during round $r > 0$, and $rec_i[0]$ be the set of n processes. Let us notice that, while it executes round r, no process knows the value of $UP[r]$, but each p_i can easily compute

the value of $rec_i[r-1]$ and $rec_i[r]$. Moreover, as crashes are "stable", p_i knows that $rec_i[r-1] \subseteq UP[r] \subseteq rec_i[r]$. The early deciding/stopping predicate $P_2 = P_{\mathsf{dif}}$ is then

$$P_{\mathsf{dif}}[i,r] \equiv \big(rec_i[r-1] = rec_i[r]\big).$$

As shown in Fig. 1, the fact that $P_{\mathsf{dif}}[i,r]$ holds does not mean that there is no crash during round r. A cross means that the corresponding process crashed during its broadcast phase, sending a message to a single process only.

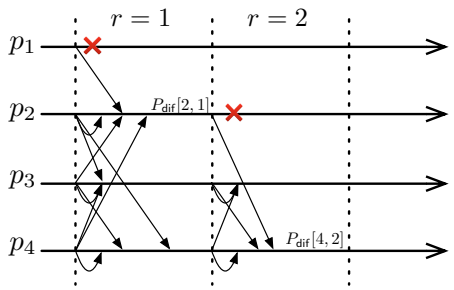

Fig. 1. An execution illustrating P_{dif}. Crosses denote crashes, $P_{\mathsf{dif}}[i,j]$ indicates when P_{dif} holds.

When $P_{\mathsf{dif}}[i,r]$ becomes satisfied, p_i received a message from all the processes that were alive at the beginning of round r. Due to the message exchange pattern, it can know all values known by these processes from the first round until the previous round $(r-1)$. Consequently, it will never know new values in the future. It follows that it can deterministically decide value among all values it know (smallest or greatest one, for example).

It is possible that $rec_i[r-1] = rec_i[r]$ while there is a process p_j such that $rec_j[r-1] \neq rec_j[r]$. As a simple example, let us consider again Fig. 1 and assume that $v_1 < \min(v_2, v_3, v_4)$ (v_i being the value proposed by p_i). During round 1, p_1 sent v_1 to p_2 only before crashing, and then, during round 2, p_2 sent v_1 to p_4 only before crashing. It follows that, while p_4 can decide v_1, no other (not crashed) process knows v_1. This issue is solved as follows: when $P_{\mathsf{dif}}[4,r]$ becomes satisfied, p_4 does not decide and stop during round r, but proceeds to round $(r+1)$ during which it broadcasts v_1 plus a flag indicating it is about to decide and stop, which it does only after the broadcast is completed.

2.3 P_{pref0} (P_3): A Knowledge-Based Unbeatable Predicate

The predicate P_{pref0}, introduced in [3], allows processes to decide as soon as possible on a preferred value, 0 in this case, while the other value 1 is decided only when the process is sure that no process decides on the preferred value 0. The predicate is expressed in the knowledge-based approach in distributed computing, in the spirit of [9]. This approach leads us to understand, in a precise sense, the information needed for a process to decide as fast as possible.

Roughly speaking, a process p_i *knows* a statement A if in every execution which is *indistinguishable* from the point of view of p_i (i.e., in which p_i has the same local view), A is true. For example, if p_i receives a message with an input 0, it knows the statement "there is a 0 in the system".

Assuming that processes want to decide as soon as possible, preferring value 0, there are two cases:

- When is it safe for a process to decide on 0? As soon as the process knows that every correct process knows that there is a 0 in the system, i.e., each correct process has received in some round a message communicating that someone started with input 0.
- When is it safe for a process to decide on 1? Since processes decide 0 as soon as possible, the process can safely decide on 1 as soon as the process knows that there is no 0 in the system, namely, no active process got a message containing a 0. Thus, no process will ever know there is a 0.

This is formalized as follows. In an execution, we say that p_j is *revealed* to p_i in round r if either p_i knows the information p_j has at the beginning of round r or it knows that p_j is crashed before that round. As a consequence, p_j cannot carry information in round r that is hidden to p_i because, in the first case, p_i knows the information p_j knows, while in the second case, p_j crashed before (hence it is not active in round r). A round r is *revealed* to p_i if every process p_j is revealed to p_i in round r. Therefore, when r is revealed to p_i, the process knows all the information than went through the system from round $r-1$ to r.

The predicate P_{pref0} is based on the following sub-predicates. Let $\exists\,\mathsf{correct\_0}(i,r)$ denote the predicate: "p_i knows that at least one correct process knows in round r that there is a 0" and let $\exists\,\mathsf{revealed}(i,r)$ denote the predicate: "a round $r' \leq r$ has been revealed to p_i". The early deciding/stopping predicate $P_3 = P_{\mathsf{pref0}}$ is defined as [3]:

$$P_{\mathsf{pref0}}[i,r] \equiv \exists\,\mathsf{correct\_0}(i,r) \vee \exists\,\mathsf{revealed}(i,r).$$

We stress that if $\exists\,\mathsf{correct\_0}(i,r)$ holds, then, at the end of round $r+1$, *every* correct process will know that there is a 0: the correct process knowing a 0 (whose existence is guaranteed by $\exists\,\mathsf{correct\_0}(i,r)$) will have communicated this value to every correct process.

The way each sub-predicate of $P_{\mathsf{pref0}}[i,r]$ is made operational will be detailed in Section 3.3, where a P_{pref0}-based algorithm is presented. To give a flavor of it, we consider below two executions, each one satisfying one sub-predicate of P_{pref0}.

- The simplest case is when a process p_i starts with input 0, then broadcasts this value to every process in round 1, and finally receives the messages sent to it in this round. At the end of the round, since p_i succinctly communicates 0 to every process, the predicate $\exists\,\mathsf{correct\_0}(i,1)$ is satisfied. Hence, using P_{pref0}, a process can decide on 0 at the end of round 1, even in presence of failures. In the execution there might be another process p_j such that $P_{\mathsf{pref0}}[j,1]$ is not true. This can happen if p_j starts the execution with input 1 and sees failures in round 1, and hence it does not decide in this round. However, p_j is prevented from deciding 1 because it knows

there is a 0 in the system (as it gets the message from p_i in round 1 containing a 0). While our example involved round 1, the same holds for an arbitrary round r: if a process broadcasts a 0 in round r and does not crash in this round, the condition $\exists\,\mathsf{correct\_0}(i, r)$ holds at the end of round r.

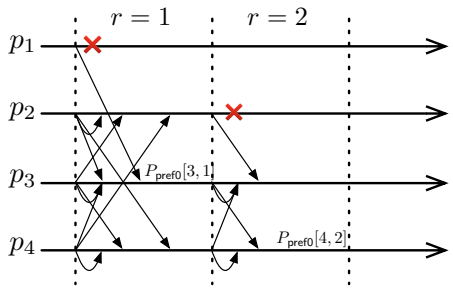

Fig. 2. An illustration of P_{pref0}. Crosses denote crashes, $P_{\mathsf{pref0}}[i, j]$ indicates when P_{pref0} holds.

– A second example is shown in Fig. 2 where every process starts with input 1. In round 1, process p_4 gets messages from every process but p_1, hence, by the end of the round, p_4 has uncertainty on the input of p_1 and the fact that this input may be known by some other process. In the example, before crashing, p_1 sends its message to p_3, and in round 2, p_4 gets a message from p_3 but not from p_2. Although p_4 sees a failure in round 2, it knows all inputs from all processes since p_4 gets indirectly the input of p_1 from the message of p_3 in round 2 (assuming full-information algorithms). Thus, round 1 is revealed to p_4 during round 2, namely, the sub-predicate $\exists\,\mathsf{revealed}(4, 2)$ is satisfied, and thus p_4 can safely decide on 1, regardless of the fact that it sees failures in both rounds.

3 Consensus Algorithms Based on the Predicates

For ease of exposition, an algorithm based on $P_2 = P_{\mathsf{dif}}$ is first presented, and only then it is shown that a simple replacement of the predicate $P_2 = P_{\mathsf{dif}}$ by $P_1 = P_{\mathsf{count}}$ produces an algorithm based on $P_1 = P_{\mathsf{count}}$. The algorithm based on $P_3 = P_{\mathsf{pref0}}$ is described at the end of the section.

3.1 An algorithm based on P_{dif} (P_2)

An early deciding/stopping consensus algorithm based on P_{dif} is described in Fig. 3. The variable r denotes the current round number, whose progress is automatically ensured by the underlying system (synchrony assumption of the distributed computing model). When the consensus algorithm starts (round 1), each process locally invokes the operation $\mathsf{propose}(v_i)$ where v_i is the value it proposes to the consensus instance. If it does not crash before, it terminates when it executes the statement $\mathsf{return}(v)$ where v is the value it decides.

Local variables. A process p_i manages three local variables.

- est_i is p_i's current estimate of the decided value. It is initialized to v_i.
- $nb_i[r]$ is the number of processes from which p_i received messages during round r. By assumption $nb_i[0] = n$. As crashes are stable, rec_i can only decrease. It follows that the predicate $rec_i[r-1] = rec_i[r]$ can be replaced by $nb_i[r-1] = nb_i[r]$.
- *early*$_i$ is a Boolean initialized to `false`. It is set to `true` when p_i discovers that it can early decide at the next round.

Local algorithm. During a round r, a process p_i first broadcasts a message carrying its current estimate est_i and the Boolean *early*$_i$ (line 3). If *early*$_i$ = `true`, p_i early decides by executing the statement return(est_i) which stops its execution (line 4). Let us notice that if p_i decides at round r, at each round $r' \leq r$, it broadcasts the smallest value it has seen up to round r'.

If p_i does not decide, it checks if another process early decides (line 5) during this round, and updates est_i according to the estimates received during the current round (line 6). Then, if its early deciding/stopping predicate is true, or if it learns another process early decides, it sets *early*$_i$ to `true` (line 8). Finally, if $r < t+1$, p_i proceeds to the next round. Otherwise, it returns its current estimate value.

```
operation propose(v_i) is
(1)   est_i ← v_i; nb_i[0] ← n; early_i ← false;
(2)   when r = 1, 2, ..., t + 1 do
      begin synchronous round
(3)       broadcast EST(est_i, early_i);
(4)       if (early_i) then return(est_i) end if;
(5)       let decide_i ← ⋁(early_j values received during current round r);
(6)       est_i ← min({est_j values received during current round r});
(7)       let nb_i[r] = number of messages received by p_i during r;
(8)       if ((nb_i[r − 1] = nb_i[r]) ∨ decide_i) then early_i ← true end if;
(9)       if (r = t + 1) then return(est_i) end if
      end synchronous round
end operation.
```

Fig. 3. P_{dif}-based early deciding/stopping synchronous consensus (code for p_i, $t < n$)

The proof of the Termination property follows directly from the synchrony assumption provided by the computing model. The proof of the Validity property follows from the observation that the est_i local variables can only contain proposed values (lines 1 and 6). The proof of the Agreement property is given in [19]. Let us notice that, in the executions where no process decides at line 4, the algorithm boils down to the very classical synchronous consensus algorithm described and proved in several textbooks (e.g., [17,19]). We prove in the following only the early decision property.

Theorem 1. *When considering the P_{dif}-based early deciding/stopping synchronous consensus algorithm, no process executes more than* $\mathsf{min}(f+2, t+1)$ *rounds.*

Proof The $(t+1)$ bound follows directly from the predicate of line 9. So let us assume that a process p_i decides at line 4 of a round d. There are two cases.

- There is a process p_i that decides at line 4 of round $d \leq f+1$. Hence, it previously broadcast the message $\text{EST}(est_i, early_i)$ at line 3, and all non-crashed processes receive this message during round d. Let p_j be any of them. If p_j does not early decide during round d, it sets $early_j$ to `true` during round d (lines 5 and 8). It follows that, if it does not crash, it will decide during the next round $d+1 \leq f+2$.
- No process decides at line 4 of a round $d \leq f+1$. Let p_i be any process that executes round $f+1$. As it did not decide by the end of the round $f+1$, we have $nb_i[r-1] \neq nb_i[r]$ at any round r, $1 \leq r \leq f$. As there are exactly f crashes, this means that we necessarily have $nb_i[0] = n$, $nb_i[1] = n-1$, ..., $nb_i[f-1] = n-(f-1)$, and $nb_i[f] = n-f$ (there is one crash per round and the process that crashed did not send a message to p_i). Moreover, as there are f crashes, we have $nb_i[f+1] = n-f$. It follows that $nb_i[f] = nb_i[f+1]$ at round $f+1$, and p_i sets $early_i$ to `true` at line 8. Hence, p_i (which is any process that executes the rounds $f+1$ and $f+2$) early decides at line 4 of round $d \leq f+2$, which concludes the proof.

$\square_{Theorem\ 1}$

3.2 An algorithm based on P_{count} (P_1)

Let us remark that $faulty_i[r] = n - nb_i[r]$. An algorithm based on P_{count} can be easily obtained from Fig. 3 by replacing at line 8 the predicate $(nb_i[r-1] = nb_i[r])$ by the predicate $P_{\mathsf{count}}[i,r] \equiv (n - nb_i[r] < r)$. The correctness proof and the bounds on the decision times of process can be proven similarly as before.

Theorem 2. *When considering the P_{count}-based early deciding/stopping synchronous consensus algorithm, no process executes more than* $\min(f+2, t+1)$ *rounds.*

3.3 An algorithm based on P_{pref0} (P_3)

Fig. 3 contains the early deciding/stopping consensus algorithm based on P_{pref0}, introduced in [3]. The processes proceed in a sequence of synchronous rounds (the variable r denotes the current round). As before, when the consensus starts, all processes simultaneously invoke the operation propose with the values they propose to the consensus instance.

Local variables. Each process p_i uses the following local variables.

- $vals_i$: set of values p_i is aware of. The set it is initialized to $\{v_i\}$ as, at the beginning of the execution, p_i knows its input only.
- G_i: directed graph containing p_i's *view* in the current round, namely, the chain of messages that has been sent to p_i so far. Initially, it has a single node $\langle i, 0\rangle$ denoting that p_i is not aware of any message before round 1. This graph is formally defined below. $V(G_i)$ denotes the vertices of G_i, and $E(G_i)$ denotes its edges.

– *knew*_0_i: Boolean indicating if p_i knows there was a zero in the previous round, namely, *vals*$_i$ contained a zero by the end of the previous round.
– *correct*_0_i: Boolean indicating if the predicate $\exists\,\mathsf{correct\_0}(i, r)$ is satisfied in the current round r.
– *revealed*$_i$: Boolean indicating if the predicate $\exists\,\mathsf{revealed\_0}(i, r)$ is satisfied in the current round r.
– *early*$_i$: Boolean indicating if p_i discovers that it can decide at the next round.

Local algorithm. At the beginning of every round, p_i first broadcasts its set of known values, *vals*$_i$, together with its view graph, G_i (i.e. it communicates all it knows so far) and then checks if it can decide early, namely, *early*$_i$ $=$ `true`. If so, it simply decides 0, otherwise, it updates its local variables, lines 5–9, in order to test the predicate $P_{\mathsf{pref0}}[i, r] \equiv \exists\,\mathsf{correct\_0}(i, r) \vee \exists\,\mathsf{revealed}(i, r)$, lines 10–14.

Before receiving the messages sent to it in the current round, p_i sets *knew*_0_i to `true` if, at the end of the previous round, p_i was aware that there was a zero in the system, line 5. Then, p_i updates its set of known values *vals*$_i$, line 6, and records in n_0_i and n_f_i the number of messages from the current round containing a zero and the number of processes it does not receive a message from in the current round, lines 7 and 8.

To explain how p_i updates its view graph G_i, let us consider the *communication graph* G_c of an execution of a full-information algorithm. Intuitively, the communication graph G_c is the directed graph that represents how communication and failures occur in a given execution. Formally, each vertex of the graph has the form $\langle i, r\rangle$, representing p_i at the beginning of round $r + 1$ (hence the vertex also represents p_i at the end of round r), and there is a directed edge $(\langle j, r\rangle, \langle i, r + 1\rangle)$ if process p_j sends a message to p_i in round $r + 1$. The *view* of p_i at the end of round r, denoted $G_c(i, r)$, is the subgraph of G_c containing every directed path that ends at $\langle i, r\rangle$. Notice that $G_c(i, r)$ contains all Lamport message chains from all processes in previous rounds to p_i at round r. Roughly speaking, $G_c(i, r)$ contains the maximal amount of information p_i has (directly or indirectly) heard of up to round r. Fig. 4 provides, as an illustration, the views of processes of the communication graph, computed for the execution of Fig. 2.

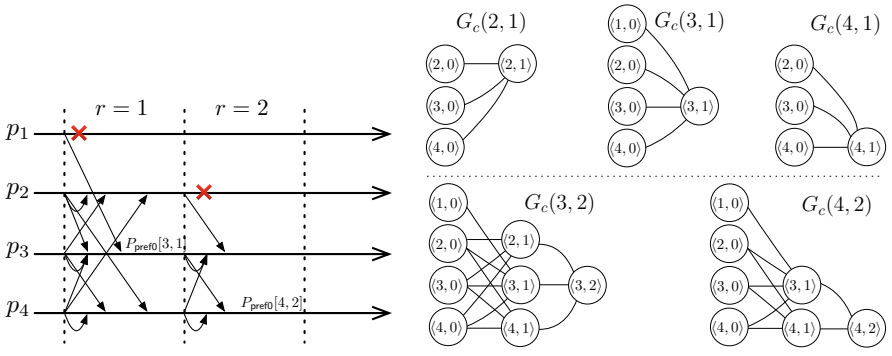

Fig. 4. Local views of the communication graph of the execution from Fig. 2

In the algorithm, p_i computes its view graph inductively as rounds go by. The main invariant in this construction is that at the beginning of round r, the local variable G_i is equal to $G_c(i, r-1)$ (which holds for $r = 1$ by the initialization of G_i). Then, at the end of round r, G_i is equal to $G_c(i, r)$ because, in line 9, p_i adds to G_i the edges due to (a) the messages it receives in round r, and (b) the view graphs of round $r-1$ carried by those messages.

Once p_i handles all messages and updates its local variables, it verifies if $P_{\mathsf{pref0}}[i, r]$ is satisfied by separately testing the sub-predicates $\exists\, \mathsf{correct\_0}(i, r)$ and $\exists\, \mathsf{revealed\_0}(i, r)$, lines 10 and 11.

operation $\mathsf{propose}(v_i)$ **is**
(1) $vals_i \leftarrow \{v_i\}$; $G_i \leftarrow (\{\langle i, 0\rangle\}, \emptyset)$;
    $early_i, knew\_0_i, correct\_0_i, revealed_i \leftarrow$ `false`;
(2) **when** $r = 1, 2, \ldots, t+1$ **do**
    begin synchronous round
(3) broadcast MSG_CONS$(vals_i, G_i)$;
(4) **if** $(early_i)$ **then** $\mathsf{return}(0)$ **end if**;
(5) **if** $(0 \in vals_i)$ **then** $knew\_0_i \leftarrow$ `true` **end if**;
(6) $vals_i \leftarrow \bigcup(vals_j$ values received during round $r)$;
(7) **let** n_0_i = number of messages received in round r with $0 \in vals_j$;
(8) **let** n_f_i = number of processes from which no message was received in round r;
(9) $G_i \leftarrow \bigcup\big(G_j$ graphs received during round r and directed edges $(\langle j, r\rangle, \langle i, r+1\rangle)\big)$;
    %% Testing $\exists\, \mathsf{correct\_0}(i, r)$
(10) **if** $(0 \in vals_i \wedge (knew\_0_i \vee (t - n\_f_i \leq n\_0_i)))$
        then $correct\_0_i \leftarrow$ `true` **end if**;
    %% Testing $\exists\, \mathsf{revealed}(i, r)$
(11) **if** $\big(\exists r' \leq r, \forall p_j, \big(\big(\langle j, r'\rangle \in V(G_i)\big) \vee \big(\exists \langle \ell, r'\rangle \in V(G_i), (\langle j, r'-1\rangle, \langle \ell, r'\rangle) \notin E(G_i)\big)\big)\big)$
        then $revealed_i \leftarrow$ `true` **end if**;
    %% Testing $P_{\mathsf{pref0}}[i, r]$
(12) **if** $(correct\_0_i)$ **then** $\mathsf{return}(0)$ **end if**;
(13) **if** $(revealed_i \wedge 0 \notin vals_i)$ **then** $\mathsf{return}(1)$ **end if**;
(14) **if** $(revealed_i \wedge 0 \in vals_i)$ **then** $early_i \leftarrow$ `true` **end if**
    end synchronous round
end operation.

Fig. 5. P_{pref0}-based early deciding/stopping synchronous consensus (code for p_i, $t < n$)

If the condition in line 10 is true, there are two not necessarily mutually exclusive subcases. If $knew\_0_i =$ `true`, then p_i knew there was a zero at the end of the previous round, hence, in the current round, it broadcasts that zero to all correct processes. And if $t - n\_f_i \leq n\_0_i$, then at least $t - n\_f_i + 1$ processes know there is a zero at the current round (where the $+1$ is because p_i itself knows there is a zero), from which follows that at least one correct process knows there is a zero, since at most another $t - n\_f_i$ processes can crash. In both cases, $\exists\, \mathsf{correct\_0}(i, r)$ is satisfied, hence $correct\_0_i$ is set accordingly.

To test if $\exists\,\mathsf{revealed\_0}(i,r)$ is satisfied, line 11, p_i directly verifies on G_i if a round is revealed to p_i: for some $r' \leq r$, for each p_j, either (a) there is a chain of messages from $\langle j, r'\rangle$ (p_j at the beginning of round $r'+1$) to $\langle i, r\rangle$ (p_i at the end of round r), i.e. $\langle j, r'\rangle \in V(G_i)$, or (b) there is a $\langle \ell, r'\rangle \in V(G_i)$ with $(\langle j, r'-1\rangle, \langle \ell, r'\rangle) \notin E(G_i)$ (i.e. p_ℓ did not receive a message from p_j in round r'). If so, the round $r'+1$ is revealed to p_i, and thus $\exists\,\mathsf{revealed\_0}(i,r)$ is satisfied.

Finally, p_i verifies if it can decide. If $\mathit{correct\_0}_i = \texttt{true}$, then all correct processes know there is a zero and hence p_i can safely decide 0, line 12. If $\mathit{revealed}_i = \texttt{true} \wedge 0 \notin \mathit{vals}_i$, then a round has been revealed to p_i and there is no zero in the system (as $0 \notin \mathit{vals}_i$), hence it is safe for p_i to decide 1, line 13. However, if $\mathit{revealed}_i = \texttt{true} \wedge 0 \in \mathit{vals}_i$, then there might be a correct process that knows a zero (but p_i does not know that fact as $\mathit{correct\_0}_i = \texttt{false}$), hence it cannot decide 1 but sets early_i to `true`, indicating that it can decide at the very next round. Observe that after p_i broadcasts its message in the next round, $\exists\,\mathsf{correct\_0}(i,r)$ is satisfied as it knew there was a zero and consequently sent its message to everyone, and thus it decides 0 in line 4.

The correctness proof of the algorithm is shown in [3]. The validity and termination properties are easy to prove. For agreement, the main observation is that the only way a process decides on 1 is if it is sure that no process ever will know there is a 0 (as it knows there is no 0 and a round has been revealed to it), hence no process will ever decide 0.

The decision time bound in the following theorem follows directly from Theorem 1 above and Theorem 4 in the next section comparing the predicates and showing that at any time that $P_{\mathsf{dif}}[i,r]$ is satisfied, $P_{\mathsf{pref0}}[i,r]$ is satisfied as well.

Theorem 3. *When considering the P_{pref0}-based early deciding/stopping synchronous consensus algorithm, no process executes more than $\min(f+2, t+1)$ rounds.*

4 Comparing the Predicates

While the three predicates presented above ensure that the processes decide in at most $\min(f+2, t+1)$ rounds in the worst cases, is one predicate better than the other?

We show here that, in a precise sense, $P_3 = P_{\mathsf{pref0}}$ is the strongest predicate for early deciding/stopping binary consensus, and $P_2 = P_{\mathsf{dif}}$ is strictly stronger than $P_1 = P_{\mathsf{count}}$, resulting in the above mentioned strict hierarchy in the sequence P_1, P_2, P_3.

Theorem 4. *Consider the predicates $P_{\mathsf{count}}[i,r]$, $P_{\mathsf{dif}}[i,r]$ and $P_{\mathsf{pref0}}[i,r]$.*

(a) Given an execution, let r be the first round at which $P_{\mathsf{dif}}[i,r]$ is satisfied. We have $P_{\mathsf{count}}[i,r] \Rightarrow P_{\mathsf{dif}}[i,r]$.
(b) Given an execution, let r be the first round at which $P_{\mathsf{pref0}}[i,r]$ is satisfied. We have $P_{\mathsf{dif}}[i,r] \Rightarrow P_{\mathsf{pred0}}[i,r]$.
(c) There are executions in which $\neg\big(P_{\mathsf{dif}}[i,r] \Rightarrow P_{\mathsf{count}}[i,r]\big)$, where r is the first round at which $P_{\mathsf{dif}}[i,r]$ is satisfied.
(d) There are executions in which $\neg\big(P_{\mathsf{pref0}}[i,r] \Rightarrow P_{\mathsf{dif}}[i,r]\big)$, where r is the first round at which $P_{\mathsf{pref0}}[i,r]$ is satisfied.

Proof Each case is handled separately.
Proof of item (a). As r is the first round during which $P_{\mathsf{count}}[i,r] \equiv (n - nb_i[r] < r)$ is satisfied, $P_{\mathsf{count}}[i,r-1]$ is false, i.e., $n - nb_i[r-1] \geq r-1$. It follows from these inequalities that $(n - nb_i[r]) - (n - nb_i[r-1]) < r - (r-1) = 1$. Combined with the fact that $nb_i[r] \geq nb_i[r]$, we obtain $nb_i[r] - nb_i[r-1] = 0$ which concludes the proof of item (a).

Proof of item (b). Since $P_{\mathsf{dif}}[i,r]$ is satisfied, we have that $nb_i[r-1] = nb_i[r]$. Therefore, in round r, p_i receives a message from any process p_j that sends a message to p_i in round $r-1$. Moreover, p_i knows for sure that all other processes crash before round r simply because it does not get any message from them in round $r-1$. We conclude that round r is revealed to p_i, from which follows that $P_{\mathsf{pref0}}[i,r]$ is satisfied (as $\exists\, \mathsf{revealed}(i,r)$ is true).

Proof of item (c). The proof follows from a counter-example. Consider a run in which $2 \leq x \leq t$ processes have crashed before taking any step, and then no other process crashes. The predicate $P_{\mathsf{count}}[i,r] \equiv (n - nb_i[r] < r)$ becomes true for the first time at round $x+1$. Let us now look at the predicate $P_{\mathsf{dif}}[i,r] \equiv (nb_i[r-1] = nb_i[r])$. We have, $nb_i[1] = nb_i[2] = n - x$. Consequently, $P_{\mathsf{dif}}[i,2]$ is satisfied. As $x \geq 2$, it follows that $\neg P_{\mathsf{count}}[i,2] \wedge P_{\mathsf{dif}}[i,2]$, which concludes the proof.

Proof of item (d). Consider any execution in which (1) all processes start with input 0, (2) p_n crashes without communicating its input to any process, and (3) all other processes are correct. Then, for every process p_i, $1 \leq i \leq n-1$, $\exists\, \mathsf{revealed}(i,1)$ is true, as p_i starts with 0 and communicates it to every one. Thus, $P_{\mathsf{pref0}}[i,1]$ is satisfied. In contrast, $P_{\mathsf{dif}}[i,r]$ is not satisfied because p_i does not receive a message from p_n, and hence $nb_i[0] = n \wedge nb_i[1] = n-1$. $\square_{Theorem\ 4}$

Operational view. The fact that $P_{\mathsf{dif}}[i,r]$ is better than $P_{\mathsf{count}}[i,r]$ comes from the following. The predicate $P_{\mathsf{count}}[i,r] \equiv (n - nb_i[r] < r)$ considers the number of crashes since the beginning, while $P_{\mathsf{dif}}[i,r]$ considers the failure pattern in a finer way: it is a differential predicate based on the number of crashes perceived by a process p_i between each pair of consecutive rounds. Similarly, $P_{\mathsf{pref0}}[i,r]$ is better than $P_{\mathsf{dif}}[i,r]$ because of the following two things: (a) each process decides on 0 as soon as possible without considering failures (as in the execution explained in the proof of Theorem 4(d)); and (b) processes detect rounds in which no information is hidden by looking at "how information flowed in the past" (like in the execution described in Section 2.3) and not only looking at the current round.

It is interesting to notice that with $P_{\mathsf{dif}}[i,r]$ (a) if no process crashes, the processes decide in two rounds, and (b) if the crashes occur before the execution, the correct processes decide in three rounds. In the failure pattern (b), $P_{\mathsf{count}}[i,r]$ does not allow to decide before round $(f+2)$. Similarly, with $P_{\mathsf{pref0}}[i,r]$, any correct process starting with 0 decides at the end of round 1, while there are executions in which such a process would decide in round $f+2$ (or, in the worst case, in round $t+1$) with $P_{\mathsf{dif}}[i,r]$.

On the unbeatability of P_{pref0}. As already mentioned, P_{pref0} is unbeatable in the sense that it cannot be *strictly* improved. Thus, there might be predicates that improve the decision time of a process in a given execution but the decision time of a (possibly different) process in a (possibly different) execution is strictly worse. An example of such a predicate is P_{pref1} where the roles of 0 and 1 are exchanged. Thus, the aim of the predicate is to decide on 1 as soon as possible (to adapt the algorithm in Fig. 5 to P_{pref1}, 0's and 1's are exchanged). Observe that in executions in which all processes start with 0, P_{pref0} is fast, regardless of the failure pattern, while P_{pref1} might need up to $t+1$ rounds, and vice versa, in executions in which all processes start with 1, P_{pref1} is fast while P_{pref0} might be slow.

Interestingly, it is shown in [15] that there is no *all case optimal* predicate P for consensus that is at least as fast as any predicate that allows to solve consensus.

A similar result was observed for the *non-blocking atomic commit* problem in synchronous systems ([8], see also Chap. 10 in [19]). According to its local computation, each process votes yes or no. If all processes vote yes and there is no failure, they all must commit their local computations. If one of them votes no, they must abort their local computations. It is shown in [8] that there is no algorithm that, whatever the decision (abort or commit), is fast in all executions: a fast algorithm for commit cannot be fast for abort, and vice versa.

5 Conclusion

This article explored the notion of early deciding/stopping for consensus, trying to better understand the relationship between static and dynamic decisions. Indeed, it turns out that dynamicity in early deciding/stopping can be based on several properties of actual execution, namely, failure pattern, flow of information, and input pattern. To compare existing solutions, we presented three early deciding/stopping strategies as a sequence of predicates, respectively based on i) counting crashed processes, ii) consecutive rounds message pattern, and iii) a finer analysis of the information flow in the execution.

On the pedagogical side, we advocate that having all algorithms presented in the same framework eases understanding and comparison of early deciding/stopping consensus algorithms, and pinpoints the subtle differences between those strategies.

The question whether such an approach can be conducted on the k-set agreement problem, the most natural extension of consensus where up to k different values can be decided [5,21], remains an open question. The predicate given in [4] is strictly better than any other predicate found in the literature but the question of its unbeatability is still an open problem.

Acknowledgements

Armando Castañeda is supported by UNAM-PAPIIT project IA102417. Yoram Moses is the Israel Pollak chair at the Technion. Michel Raynal is supported by the French ANR project DESCARTES devoted to distributed software engineering.

References

1. Aguilera M. K. and Toueg S., A simple bi-valency proof that t-resilient consensus requires $t + 1$ rounds. *Information Processing Letters*, 71:155-158 (1999)
2. Berman P., Garay J. A., and Perry K. J., Optimal early stopping in distributed consensus. *Proc. 6th Int'l Workshop on Distributed Algorithms (WDAG'92)*, Springer LNCS 647, pp. 221-237 (1992)
3. Castañeda A., Gonczarowski Y. A., and Moses Y., Unbeatable consensus. *Proc. 28th International Symposium on Distributed Computing (DISC'14)*, Springer LNCS 8784, pp. 91-106 (2014)
4. Castañeda A., Gonczarowski Y. A., and Moses Y., Unbeatable set consensus via topological and combinatorial reasoning. *Proc. 35th ACM Symposium on Principles of Distributed Computing (PODC'16)*, ACM Press, pp. 107-116 (2016)
5. Chaudhuri S., More *choices* allow more *faults:* set consensus problems in totally asynchronous systems. *Information and Computation,* 105:132-158 (1993)
6. Dolev D., Reischuk R., and Strong H. R., Early stopping in Byzantine agreement. *Journal of the ACM*, 37(4):720-741 (1990)
7. Dwork C., and Moses Y., Knowledge and common knowledge in a Byzantine environment: Crash failure *Information and Computation*, 88(2):156-186 (1990)
8. Dutta P., Guerraoui R. and Pochon B., Fast non-blocking atomic commit: an inherent trade-off. *Information Processing Letters*, 91(4):195-200 (2004)
9. Fagin R., Halpern J. Y., Moses Y., and Vardi M. Y., Reasoning about knowledge. *MIT Press* (2003)
10. Fischer M. and Lynch N., A lower bound for the time to ensure interactive consistency. *Information Processing Letters*, 14:183-186 (1982)
11. Fischer M., Lynch N. A., and Paterson M. S., Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374-382 (1985)
12. Halpern J. Y., Moses Y., and Waarts O., A characterization of eventual byzantine agreement. *SIAM Journal on Computing*, 31(3):838-865 (2001)
13. Keidar I., and Rajsbaum S., A simple proof of the uniform consensus synchronous lower bound. *Information Processing Letters*, 85(1):47-52 (2003)
14. Lamport L., Shostack R., and Pease M., The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3)-382-401 (1982)
15. Moses Y., and Tuttle M. R., Programming simultaneous actions using common knowledge. *Algorithmica*, 3:121–169 (1988)
16. Pease M., R. Shostak R., and Lamport L., Reaching agreement in the presence of faults. *Journal of the ACM*, 27:228-234 (1980)
17. Lynch N.A., *Distributed algorithms*. Morgan Kaufmann Pub., San Francisco (CA), 872 pages, ISBN 1-55860-384-4 (1996)
18. Raïpin Parvédy Ph. and Raynal M., Optimal early stopping uniform consensus in synchronous systems with process omission failures. *Proc. 16th ACM Symposium on Parallel Algorithms and Architectures (SPAA'04)*, ACM Press, pp. 302-310 (2004)
19. Raynal M., *Fault-tolerant agreement in synchronous message-passing systems*. Morgan & Claypool Publishers, 189 pages, ISBN 978-1-60845-525-6 (2010)
20. Raynal M., *Concurrent programming: algorithms, principles and foundations*. Springer, 515 pages, ISBN 978-3-642-32026-2 (2013)
21. Raynal M., *Set agreement*. 2d Edition of Springer Encyclopedia of Algorithms, pp. 1956-1959 (2016)
22. Wang X., Teo Y. M., and Cao J., A bivalency proof of the lower bound for uniform consensus. *Information Processing Letters*, 96:167-174 (2005)